Wimpy Nodes with 10GbE:
Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached

Patrick Stuedi
IBM Research, Zurich
stu@zurich.ibm.com

Animesh Trivedi
IBM Research, Zurich
atr@zurich.ibm.com

Bernard Metzler
IBM Research, Zurich
bmt@zurich.ibm.com

Abstract

Recently, various key/value stores have been proposed targeting clusters built from low-power CPUs. The typical network configuration is that the nodes in those clusters are connected using 1 Gigabit Ethernet. During the last couple of years, 10 Gigabit Ethernet has become commodity and is increasingly used within the data centers providing cloud computing services. The boost in network link speed, however, poses a challenge to the cluster nodes because filling the network link can be a CPU-intensive task. In particular for CPUs running in low-power mode, it is therefore important to spend CPU cycles used for networking as efficiently as possible. In this paper, we propose a modified Memcached architecture to leverage the one-side semantics of RDMA. We show how the modified Memcached is more CPU efficient and can serve up to 20% more GET operations than the standard Memcached implementation on low-power CPUs. While RDMA is a networking technology typically associated with specialized hardware, our solution uses soft-RDMA which runs on standard Ethernet and does not require special hardware.

1 Introduction

The ever increasing amount of data stored and processed in today's data centers poses huge challenges not only to the data processing itself but also in terms of power consumption. To be able to move from petascale computing to exascale in the near future, we need to improve the power efficiency of data centers substantially. Power consumption plays a key role in how data centers are built and where they are located. Facebook, Microsoft and Google have all recently built data centers in regions with cold climates, leveraging the cold temperature for cooling. Power consumption is also considered when choosing the hardware for data centers and supercomputers. For instance, the IBM Blue Gene supercomputer[4] is composed of energy-efficient CPUs running at a lower clock speed than traditional CPUs used in desktop computers.

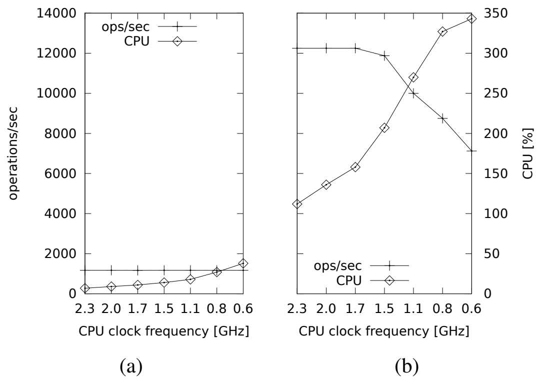


Figure 1: Performance and CPU consumption of a Memcached server attached to (a) 1GbE and (b) 10GbE for different CPU clock rates on a Intel Xeon E5345 4 core CPU

While clusters of low-power CPUs might not necessarily be the right choice for every workload [11], it has been shown that such "wimpy" nodes can indeed be very efficient for implementing key/value stores [5, 6], especially if nodes within the cluster comprise large numbers of cores. The key insight is that even a slow CPU can provide sufficient performance to implement simple PUT/GET operations, given that those operations are typically more network- than CPU-intensive. The common assumption here is that the network used to interconnect the wimpy nodes in the cluster is typically a 1 Gigabit Ethernet network. In the past couple of years, however, 10 Gigabit Ethernet has become commodity and is already used widely within data centers.

Unfortunately, the task of serving PUT/GET requests on top of a 10 Gibabit link imposes a much higher load on the host CPU, a problem for low-power CPUs running at a relatively low clock frequency. Figure 1 shows

Value Size	1K	10K	100K
Total CPU cycles	46K	84K	289K
Networking	35%	42.8%	58%
User Space	5%	3.2%	1.1%
Remaining	60%	54%	40.9%

Figure 2: Breakdown of CPU cycles used by Memcached per GET operation at 1.1Ghz CPU clock frequency

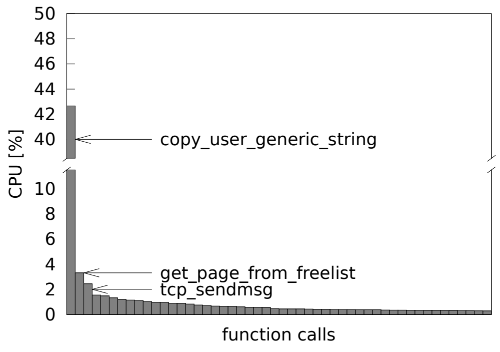


Figure 3: Distribution of the CPU consumption across function calls of the Memcached process serving 100K bytes requests

the performance and CPU consumption of a Memcached server for different CPU clock frequencies in a configuration with 1GbE and 10GbE. As the clock speed decreases in the 1GbE case, the performance stays constant, whereas the CPU consumption slightly increases. In the same experiment but using 10GbE, the performance drops significantly (still higher than what could be achieved with 1GbE though), whereas the CPU consumption reaches close to 350% (more than 3 cores fully loaded).

To get a sense of how those CPU cycles are consumed we have used OProfile [2] to divide the cycles into three classes: "network" – TCP/IP and socket processing, "application" – Memcached user space processing, and "remaining" – all other cycles executed by the operating system on behalf of the Memcached user process, e.g., context switching. Table 2 shows the cycles used by Memcached for each of those classes during the execution of a single GET operation at 1.1Ghz CPU clock frequency. For smaller size GET operations (first column, Table 2) most of the cycles are used up in the "remaining" class. For bigger value sizes (last column, Table 2), the bulk of the CPU cycles are used up inside the network stack. Figure 3 shows exactly how the network related cycles are distributed in a experiment with GET requests for 100K Bytes key/value pairs. A large fraction of the cycles are used to copy data from user space to kernel during socket read/write calls as well as during the actual data transmission.

In this paper, we propose modifications to Memcached to leverage one-sided operations in RDMA. With those modifications in place, Memcached/RDMA uses fewer CPU cycles than the unmodified Memcached due to copy avoidance, less context switching and removal of server-side request parsing. As a result, this allows for 20% more operations to be handled by Memcached per second. While RDMA is a network technology typically associated with specialized hardware, our solution uses soft-RDMA which runs on standard Ethernet and does not require special hardware.

2 Background

To provide low data access latencies, many key/value stores attempt to keep as much of their data in memory. Memcached[1] is a popular key/value store serving mostly as a cache to avoid disk I/O in large-scale web applications, thus keeping all of its data in main memory at any point in time. Remote Direct Memory Access (RDMA) is a networking concept providing CPU efficient low-latency read/write operations on remote main memory. Therefore, using RDMA principles to read, and possibly write, key/value pairs in Memcached seems to be an attractive opportunity, in particular on low-power CPUs.

Traditionally, the main showstopper for RDMA was that it requires special high-end hardware (NICs and possibly switches). However, recently several RDMA stacks implemented entirely in software have been proposed, such as SoftiWARP [15] or SoftRoCE [3]. We use the term *soft-RDMA* to refer to any RDMA stack implemented in software. While soft-RDMA cannot provide the same performance as hardware-based solutions, it has the advantage that it runs on top of standard Ethernet and requires no additional hardware support. The modifications we later propose for Memcached will be leveraging the one-sided operations of RDMA, namely *read* and *write*. Let us quickly revisit how those operations work using a soft-RDMA implementation (see Figure 4).

Soft-RDMA is typically implemented inside the kernel as a loadable kernel module. Assuming a client wants to read some data from a remote server's main memory, it will first have to allocate a receiving buffer on its own local memory. The client then registers the address and length of the receiving buffer with the in-kernel soft-RDMA provider. Similarly, the server will have to register the memory to be accessed by the client with its own local soft-RDMA provider. Registering memory with RDMA causes the RDMA provider to pin the memory and return an identifier (stag, typically 4 bytes) which can later be used by the client to perform operations on

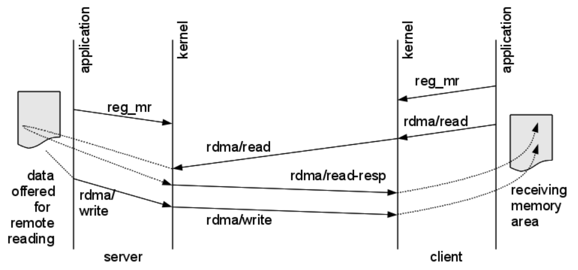


Figure 4: Example of a one-sided RDMA read operation

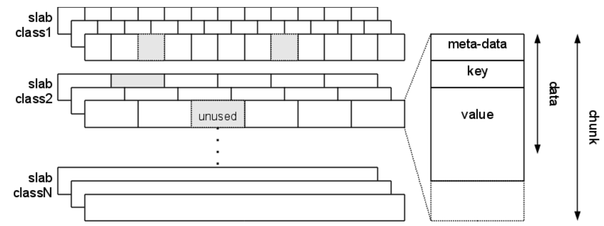


Figure 6: Memory management in Memcached

that memory. The actual data transfer is triggered by the client who issues a RDMA read operation containing the stag associated with the memory of the remote server.

Depending on which soft-RDMA stack is used, the RDMA read request may be transmitted to the server using in-kernel TCP or some proprietary transport protocol. At the server side, the read request is not passed on to the application but served directly by the in-kernel soft-RDMA stack, which – given the stag specified by the client – directly triggers transmission of the requested memory without any extra copying of the data. Copying is not avoided at the client side, but the RDMA receive path is typically shorter than the socket receive path.

In addition to read, soft-RDMA also supports write operations in a similar manner. Those operations are called one-sided since they only actively involve the client application without having to schedule any user level process at the server side. Besides avoiding the scheduling of the server process, another advantage of RDMA read/write operations is their zero copy transmit feature which yields a lower CPU consumption compared to the traditional socket based approach.

3 Memcached/RDMA

Memcached employs a multi-threaded architecture where a number of worker threads access the single hash table (Figure 5a). Serialized access to individual keys is enforced with locks. Requests entering the system will be demultiplexed to the various threads by using a dedicated dispatcher thread that monitors the server socket for new events. A key building block of Memcached is the memory management, implemented to avoid memory fragmentation and to provide fast re-use of memory blocks (see Figure 6). Memory is allocated (using malloc) in slabs of 1 MB size. A slab is assigned to a certain slab class which defines the chunk size the slab is broken into, e.g., slab classes may exists for chunk sizes of 64, 128 and 256 bytes, doubling all the way up to 1 MB. Each slab maintains a list of free chunks, and when a request (e.g., SET) comes in with a particular size, it is rounded up to the closest size class and a free chunk is taken from one of the slabs in the class. Items stored inside chunks contain key and value as well as some meta data. Chunks that are not used are put back into the free list to be re-used at a later point in time.

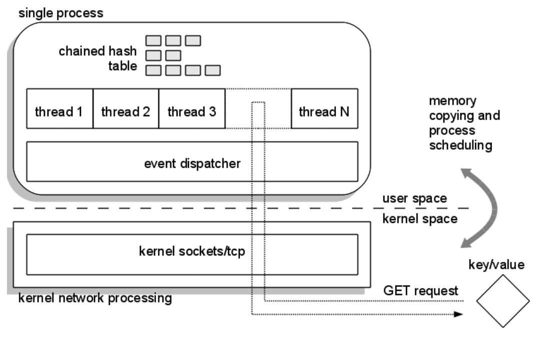
3.1 Proposed Modifications

The architecture of Memcached does contain several potential inefficiencies that can result in a significant amount of load for low-frequency CPUs. First, data is copied between the kernel and the worker threads. Given that the operations of Memcached (SET/GET) are rather simple and do not require much processing, the overhead introduced by copying data from user space to kernel and vice versa becomes significant (as shown in Figure 3). Second, in most cases processing a Memcached client request requires a context switch as the corresponding worker thread needs to be scheduled. And third, the time required to parse client requests – although small when measured in absolute numbers – can be a significant fraction of the overall processing time of a single request.

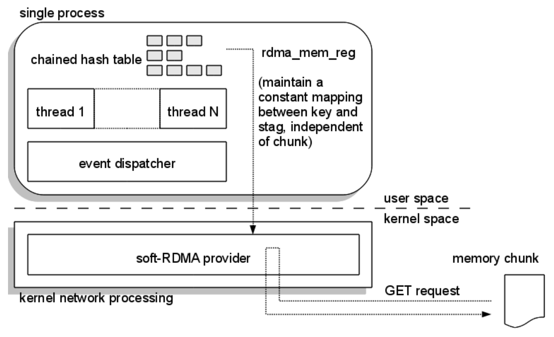
All three of those issues are addressed in our RDMA-based Memcached approach which takes advantage of RDMA's one-sided operations. In particular, we propose modifications of Memcached in the following three aspects:

1. **Memory management:** We associate memory chunks of Memcached with registered memory regions in soft-RDMA
2. **GET operation:** We use RDMA/read to implement the Memcached/GET operation.
3. **Parsing:** We increase the level of parsing necessary at the client side in favor of lowering parsing the effort at the server.

To simplify the upcoming description of the system, we henceforth use the name Memcached/RDMA when referring to the modified Memcached, and simply Memcached when referring to the unmodified system. The high-level architecture of Memcached/RDMA is shown



(a)



(b)

Figure 5: Architecture of (a) Memcached and (b) Memcached/RDMA

in Figure 5b. For a detailed understanding of the proposed modifications, let us walk through some aspects of the memory management and through the two main operations of Memcached, SET and GET:

Memory management: In Memcached, memory chunks of fixed sizes are the basic unit for storing key/value pairs. The main idea in Memcached/RDMA is to allow chunks to be read remotely without involving the Memcached user-level process. For this, Memcached/RDMA registers every newly allocated chunk with the soft-RDMA provider. The RDMA stag returned by the memory registration call is stored inside the meta data of the chunk.

SET: A client of Memcached/RDMA issues a SET request using TCP, just as if it was connected to a unmodified Memcached server. Upon receiving the SET request, the Memcached/RDMA server finds a chunk that fits the item (key/value pair) size, stores the item inside that chunk and inserts the chunk into the global hash table. If the item is a new item (the key does not yet exist in the hash table), the SET operation transmits, as part of the response message, the stag stored inside the chosen chunk. If the item replaces an already existing item with the same key, then Memcached/RDMA swaps the stags stored in those two chunks, updates the RDMA registration for both chunks, and also includes the stag of the newly created item in the response message. Clients maintain a table (stag table) matching keys with stags for a later use. *It is important to note that the swapping of stags guarantees that the same stag is used throughout the lifetime of a key, even as the chunk storing the actual key/value pair changes.*

GET: Before issuing a GET operation, the client checks whether an entry for the given key exists in the stag table. Depending on whether a stag is found or not, the GET operation is executed in the following ways:

If no entry is found, the GET operation is initiated using the TCP-based protocol. The server, however, rather than responding on the reverse channel of the TCP connection, will use a one-sided RDMA/write operation to transmit the requested key/value pair to the client using additional RDMA information that was inserted into the client request. Besides the key/value pair, the respond message from the server also includes the stag associated with the chunk storing the item at the server; this stag is inserted into the local stag table by the client.

If a stag entry is found at the client before the GET request is issued, Memcached/RDMA will use a one-sided RDMA/read operation to directly read from the server the chunk storing the requested key/value pair. The chunk, once received by the client, is parsed and the value contained in the chunk is returned to the application. The client also verifies that the key stored inside the received chunk matches the key which was requested. In case of an error the client purges the corresponding entry in the stag table and falls back to the first mode of operation.

Leveraging one-sided semantics during the GET operation has several advantages: First, data sent back to the client is always transmitted without extra copying at the server, thereby saving CPU cycles. This is true for both modes of the GET operation, namely when using RDMA/write as well as when using RDMA/read. Second, the Memcached/RDMA user level process is only involved in SET requests and in first-time GET requests by clients. All subsequent GET requests are handled exclusively by the operating system's network stack, lowering the load at the server. And third, directly reading entire chunks from the server using RDMA/read eliminates the need for parsing client requests, decreasing the CPU load at the server further.

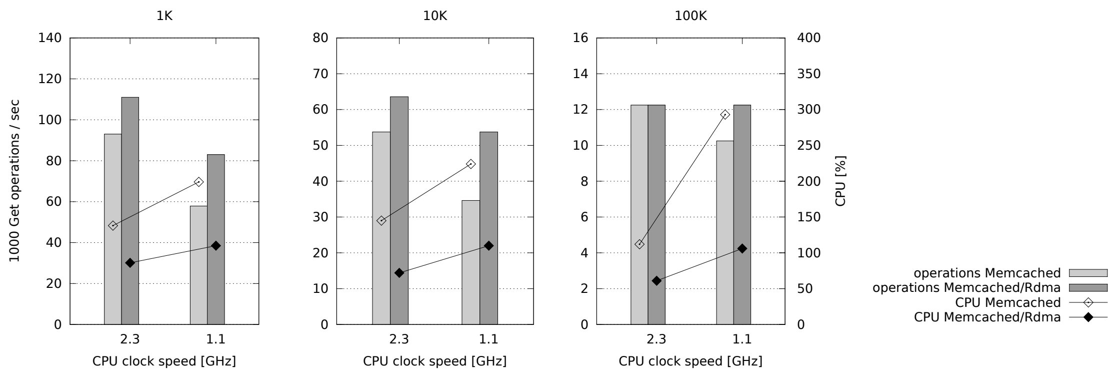


Figure 7: Performance and CPU consumption of Memcached and Memcached/RDMA for different value sizes (1K, 10K, 100K) and different server CPU clock frequencies (2.3 GHz, 1.1 GHz)

3.2 Lazy Memory Pinning

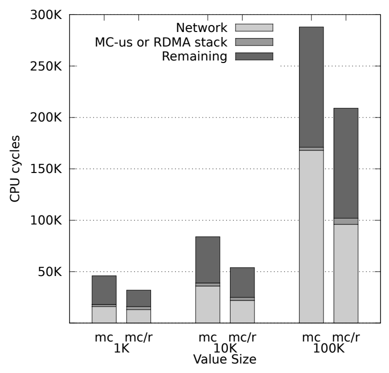
The proposed modifications, if implemented as described, can lead to a waste of memory resources if only a small fraction of the allocated chunks in Memcached/RDMA is actually read and written by the clients. As mentioned in Section 2, registering memory buffers with RDMA would typically cause the associated memory to be pinned, requiring the commitment of real physical memory. It is recommended to run Memcached with sufficient physical memory to avoid paging and the corresponding performance degradations, but overcommitting the memory resources is technically possible and supported. To maintain this feature in Memcached/RDMA, we employ a novel lazy memory pinning strategy. Any memory registered with soft-RDMA will not be pinned until it is accessed for the first time through read or write operations. Outside of RDMA operations, memory can be paged out, causing extra overhead to page in the virtual memory at the subsequent RDMA operation. If sufficient memory is available to keep all the chunks pinned, no overhead would occur. If, however, the underlying system is short of physical memory then the memory accessed by RDMA will be paged in and out according to the same OS principles any user-level memory is managed, e.g., hot memory typically stays in main memory whereas cold memory gets swapped to disk. However, we did not evaluate this feature in our experiments.

4 Evaluation

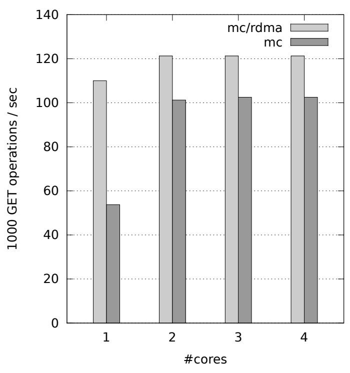
Memcached already supports both UDP and TCP transports. Thus, one appealing approach to implement Memcached/RDMA would be to just add an RDMA transport to Memcached. In this work, however, we have implemented a standalone prototype of Memcached/RDMA by re-using some of Memcached's original data structures. As one example of a soft-RDMA provider we use SoftiWARP [15], an efficient Linux-based in-kernel RDMA stack. Memcached/RDMA does not interact with SoftiWARP directly, but builds on libibverbs, which is a standardized, provider-independent RDMA API. Separating Memcached/RDMA from the actual RDMA provider will make it possible in the near future to also exploit hardware accelerated RDMA implementations (e.g., Infiniband).

Configuration: Experiments are executed on a cluster of 7 nodes, each equipped with a 4 core Intel Xeon E5345 CPU and a 10 GbE adapter. One of the nodes in the cluster acts as server running Memcached with 8 threads, whereas the other nodes are used as clients. Depending on the actual experiment, the server is configured in low-power mode, which causes the CPU to run at 1.1 GHz clock frequency. Experiments consist of two phases: in the first phase, clients insert a set of 1000 key/value pairs into Memcached; in the second phase, the clients query Memcached using GET calls at the highest possible rate. We used OProfile [2] to measure the CPU load at the server.

Performance: Figures 7 shows a performance comparison of Memcached and Memcached/RDMA in terms of number of operations executed per second at the server. Each panel illustrates an experiment for a given size of key/value pairs (1K, 10K, 100K); performance numbers are given for both 2.3 GHz and 1.1 GHz CPU clock speed. Note that the range marked by the y-axis (GET operations) differs for each panel. It is easy to see that Memcached/RDMA outperforms unmodified Memcached in almost all cases by 20% or more. The performance gap is visible for 2.3 GHz CPU clock speed and



(a)



(b)

Figure 8: (a) CPU efficiency ("mc" for Memcached, "mc/r" for Memcached/RDMA, "MC-us" for Memcached user space) and (b) performance for different numbers of cores.

increases even further once the server switches to low-power mode. An exception is the configration with 100K size key/value pairs and 2.3GHz clock speed where both Memcached and Memcached/RDMA perform equally well. The explanation for these performance results lies in the CPU consumption of Memcached and Memcached/RDMA, which we plot against the right y-axis (with 400% referring to all four cores being fully loaded) in each panel. While four cores running at 2.3GHz can handle bandwidth intensive 100K size GET requests, the difference in CPU load between Memcached and Memcached/RDMA eventually turns into a performance advantages for Memcached/RDMA at 1.1GHz clock speed.

To understand how exactly the CPU efficiency of Memcached/RDMA is materialized, we again divided the used CPU cycles into three classes "network", "application" and "remaining" (see Section 1 for descriptions of these classes). Here, the class "application" refers to the actual RDMA processing in the kernel (label "RDMA stack" in Figure 8a) since the user space Memcached process is entirely outside of the loop during GET operations. Figure 8a illustrates the used cycles per GET operation of Memcached/RDMA and compares them to the numbers we have previously shown for the unmodified Memcached (see Table 2). Clearly, Memcached/RDMA is more CPU efficient for all three sizes of key/value pairs. For smaller key/value sizes the efficiency stems partially from a low operating system involvment (e.g., context switching). For larger key/value sizes the efficiency is mostly due to zero-copy networking.

Efficiency: One obvious opportunity to increase the performance of Memcached is to throw more cores at it. This approach – besides being limited by the current scalability walls of Memcached [6, 9] – reduces the CPU efficiency of Memcached. Figure 8b illustrates that Memached/RDMA when using just a single core is able to provide a similar performance as the unmodified Memached with all four cores.

The key observation from these experiments is that Memcached/RDMA uses CPU cycles more efficiently which allows for either handling more operations per second, or for using fewer cores than a comparable unmodified Memcached server.

5 Related Work

There exists obviously a substantial body of work on key/value stores, a large part thereof adopting a disk-based storage model focusing mostly on scalability [8, 7, 14]. Fawn [5] is a key/value store designed for Flash and low-power CPUs. Like other key/value stores targetting low power [6], Fawn does not consider network speeds greater than 1GbE. Using RDMA to boost distributed storage and key/value stores has been proposed in [12, 10], and partially in [13]. These works, similar to our work, do successfully use RDMA to improve the performance of distributed systems. However, comparing the results with our work is difficult as those systems rely on dedicated RDMA hardware which is often not available in commodity datacenters. In addition, the main focus of these works is on improving throughput and latency, and not so much on reducing the CPU footprint of the system.

In contrast, our work explicitly studies high-speed networks in a low-power CPU configuration. The solution we propose leverages one-sided operations in RDMA which improves both the performance as well as the CPU consumption of the system. Furthermore, our approach can be applied within commodity data centers as it is purely implemented in software without requiring dedicated hardware support.

6 Conclusion

In this paper, we studied the feasibility of implementing an in-memory key/value store such as Memcached on a cluster of low-power CPUs interconnected with 10 Gigabit Ethernet. Our experiments revealed certain inefficiencies of Memcached when dealing with high link speeds at low CPU clock frequencies. We proposed modifications to the Memcached architecture by leveraging one-side operations in soft-RDMA. Our approach improves the CPU efficiency of Memcached due to zero-copy packet transmission, less context switching and reduced server-side request parsing. As a result, we were able to improve the GET performance of Memcached by 20%. While this paper looked at in-memory key/value stores, we believe that many of its ideas can be extended to key/value stores targeting non-volatile memory (e.g., Flash, PCM).

Acknowledgement

We thank the anonymous reviewers and our shepherd, John Carter, for their helpful comments and suggestions.

References

[1] Memcached - a distributed memory object caching system. http://memcached.org.

[2] OProfile - An Operating System Level Profiler for Linux. http://oprofile.sourceforge.net.

[3] Softroce. www.systemfabricworks.com/downloads/roce.

[4] ADIGA, N., ET AL. An overview of the bluegene/l supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference* (nov. 2002), p. 60.

[5] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 1–14.

[6] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International* (july 2011), pp. 1 –8.

[7] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[9] GUNTHER N., SUBRAMANYAM S., PARVU S. "Hidden Scalability Gotaches in Memcached" (June 2010).

[10] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (Washington, DC, USA, 2011), ICPP '11, IEEE Computer Society, pp. 743–752.

[11] LANG, W., PATEL, J. M., AND SHANKAR, S. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (New York, NY, USA, 2010), DaMoN '10, ACM, pp. 47–55.

[12] MAGOUTIS, K., ADDETIA, S., FEDOROVA, A., SELTZER, M. I., CHASE, J. S., GALLATIN, A. J., KISLEY, R., WICKREMESINGHE, R., AND GABBER, E. Structure and performance of the direct access file system. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), ATEC '02, USENIX Association, pp. 1–14.

[13] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[14] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–14.

[15] TRIVEDI, A., METZLER, B., AND STUEDI, P. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 17:1–17:5.